# Nuts and Bolts of Deploying Real Cloudproxy Applications

John Manferdelli[1], Sidarth Telang

## Overview

Cloudproxy is a software system that provides *authenticated* isolation, confidentiality and integrity of program code and data for programs even when these programs run on remote computers operated by powerful, and potentially malicious system administrators. Cloudproxy defends against observation or modification of program keys, program code and program data by persons (including system administrators), other programs or networking infrastructure. In the case of the cloud computing model, we would describe this as protection from co-tenants and data center insiders. To achieve this, Cloudproxy uses two components: a "Host System" (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to the protected program or "Hosted System" (VM, Application, Container).

This document focuses on installation, security and deployment tradeoffs for Cloudproxy applications. In addition, we explain at length several options for providing key management and program operation continuity as programs are upgraded or new programs are introduced. Programmers new to Cloudproxy often worry that program upgrade is impossible or difficult but it is not. We also describe successful key management strategies which support frequent key rotation which is strongly recommended to ensure ongoing security as standard practice as well as providing resilience in the face of errors or omissions.

We hope these instructions allow safe and rapid installation and configuration of Cloudproxy nodes without extensive training or preparation. We assume readers are familiar with [4] which explains Cloudproxy basic concept like measurement and principal names and explains how to build Cloudproxy applications.

Readers can consult [1] for a fuller description. Source code for Cloudproxy as well as all the samples and documentation referenced here is in [2].

## Downloading and compiling Cloudproxy

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

    git clone https://github.com/jlmucb/cloudproxy,

or,

    go get https://github.com/jlmucb/cloudproxy.

---

[1] John is manferdelli@google.com or jlmucbmath@gmail.com. Cloudproxy is based on work with Tom Roeder (tmroeder@google.com) Sid Telang (sidtelang@google.com), Fred Schneider (Cornell) and Kevin Walsh(kevin.walsh@holycross.edu).

This latter command will also install the needed go libraries. You can also download a zipped repository from github. You should probably install this in ~/src/github.com/jlmucb (which we refer to as `$CLOUDPROXYDIR`) to save go compilation problems later. It's a good idea to put go binaries in `~/bin` as is common in Go. Follow the installation instructions in `$CLOUDPROXYDIR/Doc`; that directory also contains [1] and an up to date version of [4].

You must also install the Go development tools (and C++ development tools if you use the C++ version) as well as protobuf, gtest and gflags as described in the Go documentation.

We will continue to use the simpleexample application described in [4].

## Security Model

Any deployment of a security system must consider the security or threat model that a system seeks to provide. For the purpose of this Deployment Guide, we consider you are running software for your clients in a cloud data center (or other shared facility) and storing your data there.

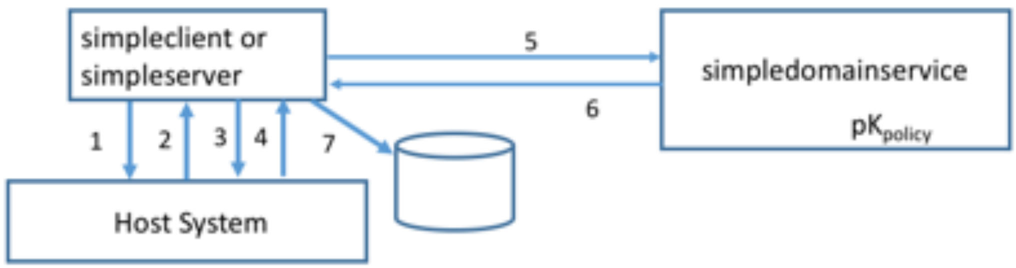
Your goal is to provide the principled, verifiable *confidentiality and integrity* of the execution of your programs and the data they read and store from unrestricted attacks from outsiders (e.g. – other clients of the cloud provider) as well as attacks from individual insiders in the cloud data center.

We do assume that data center insiders do not have physical access to computers running your software while that software is running. If they did, insiders could mount physical attacks on running hardware, for example, attaching memory probes to buses. This level of protection can be ensured by, say, enclosing the computers running client software in locked cages that prevent physical access to designated computers while the computers are running Cloudproxy, and for a few minutes afterwards (to protect against insiders harvesting memory remnants of running programs). An extra measure to ensure compliance might be cameras monitoring the caged systems. We *do not* require all computers or networking equipment to be in such cages. In fact, storage systems and networking equipment could remain unprotected without violating the foregoing security guarantee.

We also require that the software you provide (or use) not have remotely exploitable vulnerabilities but we *do not* limit what software you can run or significantly change the common programming model.

## Overview of installation on Linux

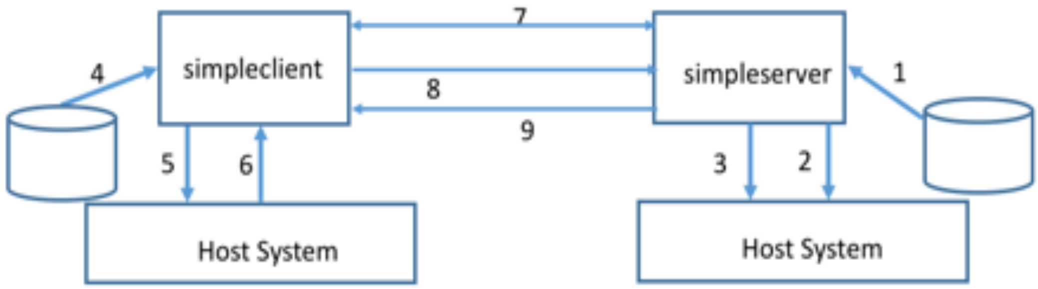
The figure below should be familiar from [4]. We will refer to this throughout the document.




**Initialization**

1. simpleclient (or simpleserver) generates public/private key pair $PK_{simpleclient}$, $pK_{simpleclient}$. simpleclient requests Host System attest $PK_{simpleclient}$.
2. Host System retuns attestation
3. simpleclient generates additional symmetric keys and request Host System seal symmetric keys and $pK_{simpleclient}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpledomainservice and transmits attestation.
6. simpledomainservice returns signed Program Certificate.
7. simpleclient stores sealed blobs and Program Certificate for later activations.

**Initialization**




Operation
1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.

4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.

7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

**Operation**

All the hardware enforced mechanisms described in this document are based on the primary system software being Linux based. This includes a native Linux booted on an Intel SMX/TPM platform as well as the KVM hypervisor. Two configurations we discuss (a hosted Linux stacked on KVM and a Docker based Tao stacked on a Linux Tao) do not run directly on hardware but are hosted by a lower level Linux based Tao. We refer to a Cloudproxy Host booted on hardware as a Base CloudProxy Host. The comments in this section thus refer to a KVM base system or a native Linux Base CloudProxy system. With fairly minor changes, one could also make an BSD based platform a Base CloudProxy Host.

The need to perform a measured requires a little more software support. Linux Base CloudProxy Host use a multi-boot sequence to load the Linux kernel mediated by Grub. The stack of booting software consists of:

1. Grub which loads the other booting software into memory.
2. Tboot which mediates the authenticated boot using SMX and the TPM.
3. The Linux kernel with TPM support.
4. A custom initramfs which is measured as part of the boot and contains the compiled Cloudproxy components and critical libraries. Unlike the normal boot sequence, the initram filesystem remains the root file system throughout the base system's activation. All otherfile systems are mounted under it.

5. Dmcrypt which is used to encrypt the pagefile and swap space[2] (so no unencrypted data is stored on the paging disk).

We describe the role and general installation procedure for each of these below. For each, there is also an appendix with detailed installation instructions. In addition, we describe preparing, measuring and running Linux based VM's with Cloudproxy support which can be hosted by KVM.

The overall procedure for preparing and installing Cloudproxy root system is:

1. Install Linux. TPM2.0 requires a Version 4 kernel or later. The kernel should contain the appropriate tpm driver and dmcrypt. We remark that, for security reasons, the TCB of the root CloudProxy system should be as small as possible. A carefully constrained kernel is a critical element of security. For example, our Linux root CloudProxy systems either disallow loading dynamic modules or subject proposed loads to a small whitelist of signed modules. The root CloudProxy should include Dmcrypt.
2. Install the Linux Kernel Development environment, on development machines.
3. Enable SMX, IOMMU and VT extensions in BIOS. Activate the TPM using the BIOS in the case of TPM 2.0 or using trousers in the case of TPM 1.2.
4. Take ownership of the TPM and generate the appropriate keys.
5. Build TBOOT. Retrieve the required Authenticated Code Module (ACM) required by Tboot and place it and the tboot image in /boot as described below.
6. Build the Cloudproxy binaries. Then build a custom initramfs with these binaries as described below. Put the newly created initramfs in /boot/
7. Configure Grub. You will boot a grub configuration which names Tboot, the ACM, your Linux kernel and your custom Initramfs.
8. Boot!

The "measurement" of a TPM mediated boot will be in PCR 17 and 18 of your TPM. You will need this measurement for your domain service and the easiest way to get it is to have Cloudproxy report it after it boots on your test hardware. Cloudproxy does the rest! After you have successfully carried out these steps, you can build applications as described in [4].

Next, we describe each of these steps in more detail. After that, we discuss secure, scalable deployment of Cloudproxy applications including key management techniques followed by a brief description of Security Considerations which should be thought through for your particular deployment needs.

---

[2] A subtle point: Dmcrypt at the time of this writing, encrypts but does not provide integrity protection. As a result, the "paging disk" (which might be flash or spinning media) should be in the "physical protection barrier" described in the Security Considerations section of this document.

**Hardware root of trust**

Because of a lack of physical control over the hardware (and even with physical control in the face of errors), maintaining our security model requires a hardware root of trust that is used to measure the base system software that boots and ensure that only software and only software trusted by you can access some basic system secrets like signing keys. Without this, insiders could boot any privileged software and violate the security model you seek to enforce. Cloudproxy, as mentioned above can employ TPM1.2 or TPM 2.0 as “off the shelf” mechanisms. So first you must enable and provision one of these.

**BIOS**

In Bios, you must ensure that VT and IOMMU support is enabled. The Tpm must also be enabled (usually it is enabled by default in the case of TPM 2.0 but is disabled by default with TPM 1.2).

**TPM 1.2 Installation**

install trousers

    sudo agt-get install trousers
    sudo agt-get install tpm-dev
    sudo agt-get install tpm-tools
    sudo agt-get install libtspi-dev

These packages have the trousers utilities we will use to initialize the TPM.

Enable and activate the TPM in BIOS and make sure TXT and vt-d is enabled too. After enabling the TPM in BIOS, type:

```
sudo bash
  tscd start
  tpm_takeownership -z -y
  tpm_getpubek
```

If someone has already taken ownership, you must disable the tpm and reenable it. You may have to remove power to reset the TPM and you may have to create an endorsement key using tpm_createek

You can activate the TPM from software (trousers) if you are in single user mode. The sequence of instructions is:

```
telinit S or boot to single user mode
ifup lo
start tcsd
tpm_setpresence --assert
tpm_setenable --enable --force
tpm_setactive --activate --force
```

```
reboot
```

First we must initialize the TPM physical chip with the tpm_clear command, which returns the TPM to the default state, which is unowned, disabled and inactive. That command wipes all the ownership information from the TPM, invalidates all the keys and data tied to the TPM and even disables and deactivates the TPM.

**TPM 2.0 Installation**

Unlike TPM 1.2, there is no need to download trousers or other utilities. Taking ownership, and enabling TPM 2.0 is all done in the BIOS interface and all our TPM 2.0 support is self-contained in Cloudproxy components.

**Tboot**

Tboot coordinates the TPM measured boot with CPU and chipset support to carry out the measured boot, attestation, sealing and unsealing required by the root Cloudproxy Tao.

Tboot requires an ACM Module that sets up the machine for TXT. The appropriate SINIT Module can be downloaded from http://software.intel.com/en-us/articles/intel-trusted-execution-technology/.

The source code for Tboot itself should be downloaded from https://sourceforge.net/projects/tboot/ and copied into /boot.

It is relatively easy to build. In the source directory, say

```
make
```

And then as root

```
make install
```

Tboot will show the appropriate device ID in its output if it is booted with the incorrect version of sinit. You can use the tboot utility txt-stat (as root) to read the most recent tboot logs. The current version of tboot requires version 17 or greater of the SINIT module. A copy of the module we used is in this distribution.

For TPM 1.2 you will need to incorporate tboot into you tpm 1.2 launch policy.

Detailed copy and paste instructions are in the appendix.

**InitRamfs**

Everything upon which the security of your software relies must be measured, including basic system software and libraries you rely on. On Linux, we can protect and measure all that requires protection by incorporating it into a small, in memory, filesystem provided with the kernel boot image.

Initram is an in memory file system used by the Linux kernel. In most cases, initramfs is discarded late in the boot sequence of Linux in favor of the “real” mounted system disk but in our case, initram remains the root file system throughout the Base Linux Cloudproxy lifetime (other file systems are mounted under it). We modify “stock” initramfs in several ways. First, we get rid of any unnecessary functionality. Next we include the Cloudproxy components and, in some cases, even the Cloudproxy applications are included in initramfs. Lastly, we modify the stock scripts so that the “disk based” root file system is mounted under the initramfs rather than replacing it.

Detailed copy and paste instructions for initramfs are in the appendix.

**Dmcrypt**

Since most Linux systems employ virtual memory systems which temporarily store memory pages on disk, we must prevent secrets from being saved to a disk in a way that could be modified as programs are running or read after programs stop running. This would violate our security model. To do this, we employ Dmcrypt which encrypts the disk used for paging. We also assume this disk is contained within the cage housing the Cloudproxy programs (since Dmcrypt does not provide integrity protection). However, we *do not* require that the paging disk be protected before or after being used.

Detailed copy and paste instructions for configuring Linux to use dmcrypt are in the appendix.

**Grub configuration**

Grub is used to load the ACM, tboot, the OS (or hypervisor) and initramfs into main memory so that the measured boot can be carried out. Note that one can employ a grub configuration that can boot non Cloudproxy enabled images without harm.

Detailed copy and paste instructions for grub setup are in the appendix.

## Installing and configuring a Cloudproxy supported KVM instance

We use KVM as the Cloudproxy hypervisor that can host isolated, measured Cloudproxy VMs.

Detailed copy and paste instructions for KVM setup are in the appendix.

## Installing and configuring a Cloudproxy supported TPM hosted Linux

In addition to a hypervisor rooted Cloudproxy stack, one could simply boot a Cloudproxy enabled Linux.

Detailed copy and paste instructions for Linux root setup are in the appendix.

## Installing and configuring a Linux stacked on KVM

When using a hypervisor rooted Cloudproxy system, we need to provide Cloudproxy enabled VMs. Cloudproxy can support "stacked" systems that extend root protection to any layer of a software stack.

Detailed copy and paste instructions for configuring a stacked Linux Tao are in the appendix.

## Installing and configuring Docker containers

Another stacked system protected by Cloudproxy is a Docker container within a Linux host (or stacked Linux host).

Detailed copy and paste instructions for configuring and running a stacked Docker Tao are in the appendix.

### Host Certificates

Your TPM must have a signed certificate for its public. Consult the final appendix to provision this certificate. The underlying key is called an AIK in the case of TPM 1.2 and an Endorsement Certificate in the case of TPM 2.0. In many cases, an Endorsement Certificate is included in NVRam for TPM 2.0.

## Domains

What's a domain? Where is domain information. How is it initialized? How is it used at runtime. Cover policy key creation and distribution.

## Key management

Key management supports the storage, distribution and critically, the rotation of symmetric content keys as well as establishing the "Program Key" for Cloudproxy programs (and hosts) and rotating the Program Key as keys expire or cloudproxy programs are changed.

In most of our programs, keys and content types (like files) are objects and have domain-wide hierarchical universal names and epochs. Epochs are monotonically increased. For example, a file used by several Cloudproxy programs may contains customer information like customer names, addresses, etc. Since these are stored objects, they are encrypted and integrity protected and typically stored redundantly at several network accessible locations. The file

/jlm/file/us-zone/customer-file-location1, epoch 2, type: file refers to the file with the indicated name; the epoch indicates the version of the file, different versions of the file are usually encrypted with different keys. These encryption keys also have domain-wide hierarchical universal names and epochs. Encryption keys protect other objects like keys or files. To decrypt a file, we need a chain of keys starting at the sealing key for the Cloudproxy program on a given node and terminating in the desired object (say a file). Each node of the chain consists of the name of the protector object, the protected objects and an encrypted blob consisting of the protected object encrypted (and integrity protected) by the protector key.

For example

/jlm/program-name/master-sealing-key, epoch 5, type: aes-128-gcm
/jlm/key/us-zone/customer-folder-keys, epoch 2, type: aes-128-gcm
/jlm/key/us-zone/customer-file-key, epoch 3, type: aes-128-gcm
/jlm/file/us-zone/customer-file-location1, epoch 2, type: file

Often domains will want to partition keys to provide resilience by *protection zones.* One common way to do this is to have different keys for different geographic locations. This is easily accomplished by reserving a level of the hierarchical object namespace for the zone name as is done above.

Although the foregoing name, epoch based mechanisms facilitate key rotation, it is not a complete solution. Let's first consider, given this framework, how we might rotate keys.

There are two common circumstances where new keys are introduced into a Cloudproxy environment. First, keys should be rotated regularly as part of good cryptographic hygiene and second, when programs change, their measurements change which means they can no longer access sealed data for earlier versions of the program. In both cases, similar techniques can be used to carry out the key rotation.

<u>Key Rotation and software upgrades</u>

See go/support/support_libraries /protected-objects and go/support/support_libraries/rotation_support for support routines to maintain and encrypt content keys and build chains of keys like:

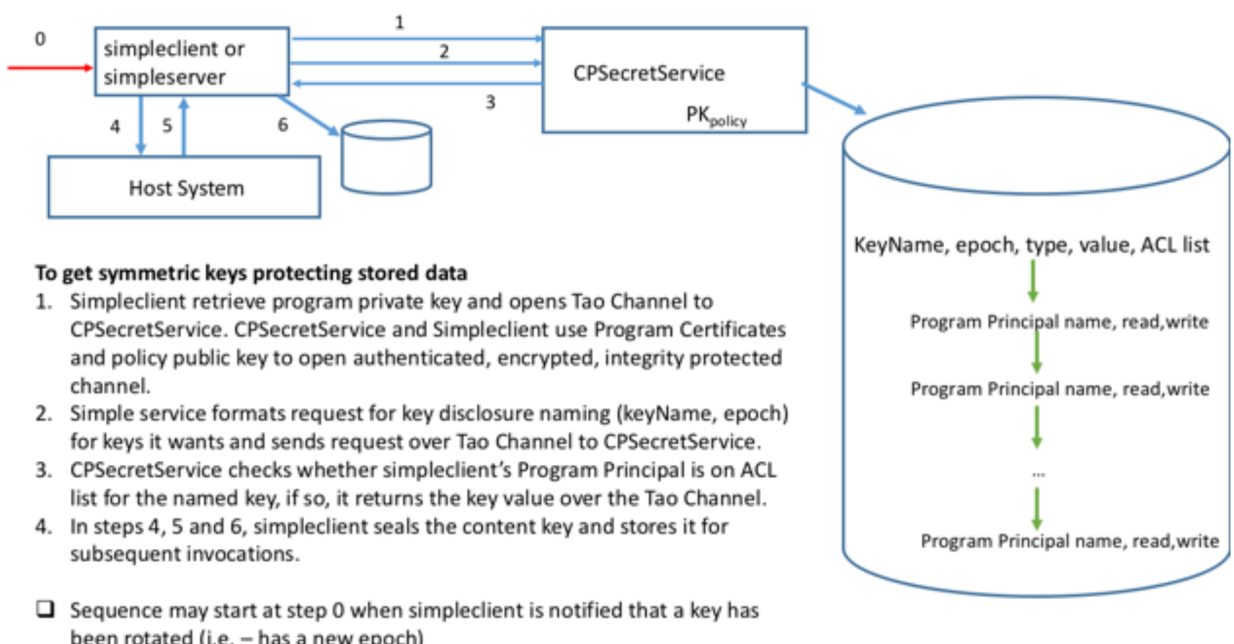
> Protector Object: /jlm/program-name/master-sealing-key, epoch 5, type: aes-128-gcm
> Protected Object: /jlm/key/us-zone-key, epoch 2, type: aes-128-gcm
> Encrypted Protected Object
>
> Protector Object: /jlm/key/us-zone-key, epoch 2
> Protected Object: /jlm/key/us-zone-key/customer-folder-key, epoch 3, type: aes-128-gcm
> Encrypted Protected Object

Protector Object: /jlm/key/us-zone-key/ customer-folder-key, epoch 3
Protected Object: /jlm/key/us-zone-key/customer-folder-key/customer-file-key, epoch 2, type: aes-128-gcm
Encrypted Protected Object

Protector Object: /jlm/key/us-zone/customer-file-key, epoch 3, type: aes-128-gcm
Protected Object: /jlm/file/us-zone/customer-file-location1, epoch 2, type: file

*Using a keystore*

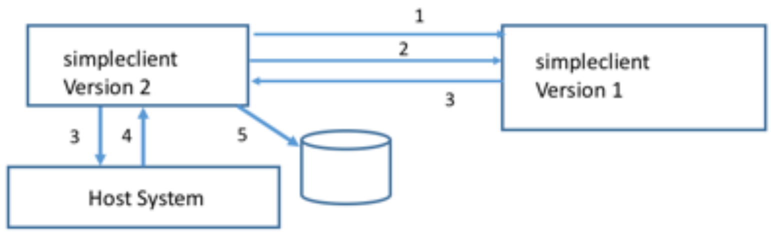
A sample keystore with the required functionality is implemented in go/support/infrastructure_support/CPSecretServer. Needless to say, this program itself would likely be implemented as a cloudproxy program.



**Getting content keys**

Since keys can be fetched any time after startup and no pre-existing state is required (except for the Program key), key rotation is easy and is focused at the Secret Service. The secret service can also publish alerts when new keys are available. Upgrade when program versions change is also easy. The new version of the program just gets keys from the server. The server is updated with new ACLs as new program versions become available.

*Using policy key disclosure*




**To get symmetric keys protecting stored data from another program or version**

1. Simpleclient, version 2 retrieves program private key and opens Tao Channel to Simpleclient, version 2. Simpleclient, version 2 and Simpleclient, version 1 use Program Certificates and policy public key to open authenticated, encrypted, integrity protected channel.
2. Simple service formats request for key disclosure naming (keyName, epoch) for keys it wants and sends request over Tao Channel to CPSecretService accompanied with policy-key signed certificate stating policykey says ProgramPrincipalName (simpleclient, version 2) can read /jlm/zone-us/customer-folder-key, epoch 2.
3. Simpleclient, version 1 transmits keys over tao channel.
4. 4, 5, 6, Simpleclient, version 2 seals retrieved key for subsequent use.

**Getting content keys**

See go/support/support_libraries/secret_disclosure_support for sample code to construct and verify a policy-key signed secret-disclosure statement.

Policy-key says PrincipalName can-read /jlm/key/us-zone/customer-folder-key, epoch 2

*A note on key rotation*

Either the keystore mechanism or the secret disclosure mechanism can be used to protect other keys.  For example, they can help disclose private signing keys to controlled groups.  This allows a program to authenticate itself as a group or a standard Linux “service accout,” making interoperability with “legacy” authorization systems easier.

## Scale and deployment considerations

### Just write your applications properly:  simple, right?

While the Cloudproxy security model makes security reliance transparent and easily manageable, you still have to write your programs so that adversaries cannot exploit flaws in the programs you write (or the author of the VMM or BIOS wrote).  This is not a trivial task but more programming tools, better programming languages and new techniques (such as proof-carrying-code) present plausible models for “much safer” software you can trust.

### Oops: What to do if there’s a gigantic breach

Although it is hopefully a rare event, a large scale failure (maybe caused by flaws in critical software) can be remedied simply by redistributing the application with a new policy key (after fixing the security flaws in this case).

## Redundancy, failover and avoiding universal breaches

## Planned features

Local state rollback support, a few more crypto algs

## Out of town, call collect

Bugs, suggestions

## Acknowledgements

Tom Roder, Fred Schneider, Kevin Walsh.

## References

**[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing,** http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
**[2] CloudProxy Source code,** http://github.com/jlmucb/cloudproxy. Kevin Walsh and Tom Roeder were principal authors of the Go version.
**[3] TCG, TPM specs,** http://www.trustedcomputinggroup.org/resources/tpm_library_specification
**[4] Beekman, Manferdelli, Wagner,** Attestation Transparency: Building secure Internet services for legacy clients**.** AsiaCCS, 2016.
**[5]** Manferdelli, CloudProxy Nuts and Bolts.
**[6] Intel,** The MLE Development Guide

# Appendix 1 - Supported Hardware

Intel or AMD CPU's and chipsets with secure extensions that support SKINIT.  In addition, there must be a physical TPM on the motherboard or in firmware.  Most new Intel CPU's have firmware TPM's so most new hardware is "automatically" Cloudproxy enabled.

## TPM 1.2 Initialization and Operation

# tpm_clear --force
Tspi_TPM_ClearOwner failed: 0x00000007 - layer=tpm, code=0007 (7), TPM is disabled
We can see that the TPM is disabled, which is why we can't clear it. This can happen if we forget to actually enable the TPM in BIOS. The first thing to do would be to actually enable the TPM in BIOS. But if the TPM has been initialized before, we would receive the output that can be seen below:

# tpm_clear --force
TPM Successfully Cleared.  You need to reboot to complete this operation.  After reboot the TPM will be in the default state: unowned, disabled and inactive.
This would require us to reboot the computer for changes to take effect. When clearing the TPM we'll return it to the default state, which is unowned, disabled and inactive, as already mentioned. To enable the TPM afterwards, we need the owner password. But since the TPM owner has been cleared, there is no owner password and we can set a new one without entering the old one. We can also receive an error like the following:

# tpm_clear --force
Tspi_TPM_ClearOwner failed: 0x0000002d - layer=tpm, code=002d (45), Bad physical presence value

DO NOT SET ANY PASSWORD for the TPM.
# tpm_takeownership -z -y
If we later want to change either of the commands, we can do it with the tpm_changeownerauth command. If we pass the –owner argument to the tpm_changeownerauth command we'll be changing the administration password and if we pass the –srk into the tpm_changeownerauth command we'll be changing the SRK password. We can see the example of both commands in the output below:

# tpm_changeownerauth --owner
Enter owner password:
Enter new owner password:
Confirm password:

# tpm_changeownerauth --srk
Enter owner password:
Enter new SRK password:
Confirm password:

There are 5 keys in TPM:

TPM Endorsement Key (EK): This key is created by the manufacturer and cannot be removed. Sometimes it can be changed by the owner of the computer.

TPM Storage Key (SRK): Is the 2048 bit RSA key created when configuring the ownership. This key is stored inside the chip and can be removed. The key is used to encrypt the Storage Key (SK) and Attestation Identity Key (AIK).# tpm_setenable --enable

Enter owner password:

Disabled status: false

# tpm_setactive

Enter owner password:

Persistent Deactivated Status: false

Volatile Deactivated Status: false

There are usually two Endorsement Keys (EK): the public and private one. The private key is always stored at the TPM and cannot even be seen by anyone, while the public key can be displayed with the tpm_getpubek command.

# tpm_getpubek

Tspi_TPM_GetPubEndorsementKey failed: 0x00000008 - layer=tpm, code=0008 (8), The TPM target command has been disabled

Enter owner password:

```
Public Endorsement Key:
  Version:   01010000
  Usage:     0x0002 (Unknown)
  Flags:     0x00000000 (!VOLATILE, !MIGRATABLE, !REDIRECTION)
  AuthUsage: 0x00 (Never)
  Algorithm:         0x00000020 (Unknown)
  Encryption Scheme: 0x00000012 (Unknown)
  Signature Scheme:  0x00000010 (Unknown)
  Public Key:
        a350b3a3 3edddc30 06248f4f 5d3eb80a 34fcbea0 83dde002 8dffa703 e116f8b0
        eb1962ee a65998b3 384aeb6e 85486be9 0316a6ca a189a5ba 2217b2a2 9da014db
        dfbe7731 fb675e7a 438c4775 deea54fb 0c75de5d ba961950 3eda4555 d27a9a30
        e94d39d0 a4ea314d a70eaf08 e49dd354 d57ed34d 234220d9 604471a9 86173050
        9ff9b0e5 b65cb4b5 5f46a7f9 4378bd7e 8c61b91b ad312974 fef5d70f 84f4484f
        e5c95300 0eef76f2 1667443f dc2fa82e 351d945e 6b5f75e8 828d010f 61541552
   [...]
```

# Tboot Launch Policy for TPM 1.2

You may build a Launch Control Policy (LCP) which controls what can be booted. If you have no LCP, the default policy will be applied and this should work! If not, there are instructions on creating policy at the bottonm of this file. Latest tboot test was tboot-1.7.3.2 with SINIT67.BIN.

Trousers and trousers-devel packages must already be installed in order to build lcptools.

Next you must modify grub so that you can choose the tbooted linux from the boot window.

If the verified launch policy is non-empty, it is extended into PCR 17 by default. Subsequent loaded modules are extended into PCR 18 by default (Check this!). This behavior can be changed with the VLP.

The Grub(2) configuration file is usually in /boot/grub and is called grub.cfg. It is updated automatically when a kernel is updated or when you run update-grub.

Tboot module must be added as the 'kernel' in the grub.conf file.

Note that the Lenovo T410 is known not to work with the Intel IOMMU; it will not successfully boot with TBOOT.

The final grub configuration file will look something like:

```
menuentry 'Ubuntu Linux 3.0.0-16-generic with TXT' --class ubuntu --class gnu-linux --
class gnu --class os {
        recordfail
        set gfxpayload=text
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos5)'
        search --no-floppy --fs-uuid --set=root 8ab78657-8561-4fa8-af57-bff736275cc6
        echo 'Multiboot'
        multiboot /boot/tboot.gz /boot/tboot.gz logging=vga,memory,serial
        echo 'Linux'
        module   /boot/vmlinuz-3.0.0-16-generic /boot/vmlinuz-3.0.0-16-generic
root=UUID=8ab78657-8561-4fa8-af57-bff736275cc6 ro   splash vt.handoff=7 intel_iommu=on
        echo 'initrd'
        module /boot/initrd.img-3.0.0-16-generic /boot/initrd.img-3.0.0-16-generic
        echo 'sinit'
        module /boot/sinit51.bin /boot/sinit51.bin
```

Modify grub.conf to load the policy data file:

Edit grub.conf and add the following:

    module /list.data

   where you should use the path to this file.
Copy sinit into /boot and change run grub.conf then run update-grub.
Check the /boot directory to make sure tboot.gz is there.

The utility txt-stat in the utils subdirectory of tboot, can be used to view the DRTM boot log.

Reported problems with TPM's

The TPM, driver/char/tpm/tpm.c, depends on TPM chip to report timeout values for timeout_a, b, and d. The Atmel TPM in Dell latitude 6430u, reports wrong timeout values (10 ms each), instead of TCG specified (750ms, 2000ms, 750ms, 750ms for timeout_a/b/c/d respectively). You can fix the driver code and it will work.

A permissive Tboot policy is required for the non-hypervisor solutions. In this case, the system will boot whatever it finds, but it will still perform measured launch, and clients interacting with the system will still be able to check that the right software was booted by checking the PCRs. Once the hypervisor is written, a more restrictive launch control policy is possible and maybe desirable.

Put 11_tboot in /etc/grub.d

Information on PCR Usage on TPM 1.2

PCR 17 will be extended with the following values (in this order):
- The values as documented in the MLE Developers Manual
- SHA-1 hash of: tboot policy control value (4 bytes) SHA-1 hash of tboot policy (20 bytes) : where the hash of the tboot policy will be 0s if TB_POLCTL_EXTEND_PCR17 is clear

PCR 18 It will be extended with the following values (in this order):
- SHA-1 hash of tboot (as calculated by lcp_mlehash)
- SHA-1 hash of first module in grub.conf (e.g. Xen or Linux kernel)

PCR * : tboot policy may specify modules' measurements to be extended into PCRs specified in the policy
The default tboot policy will extend, in order, the SHA-1 hashes of all modules (other than 0) into PCR 19.

The LCP consists of policy elements.

To specify launch policy via a list of hashes:
```
0.  sudo bash
1.  lcp_mlehash -c "command line for tboot from grub.cfg" /boot/tboot.gz > mle_hash
  - the command line in this case is the string from /boot/grub/grub.cfg
  - after multiboot tboot.gz, e.g., "logging=vga,memory,serial"
  - copy and paste:
./lcp_mlehash -c "logging=vga" /boot/tboot.gz>mle_hash
2.  lcp_crtpolelt --create --type mle --ctrl 0x00 --minver 17 --out mle.elt mle_hash
  - copy and paste
./lcp_crtpolelt --create --type mle --ctrl 0x00 --minver 17 --out mle.elt mle_hash
```

```
3.  lcp_crtpollist --create --out list_unsig.lst mle.elt pconf.elt
  - copy and paste
./lcp_crtpollist --create --out list_unsig.lst mle.elt pconf.elt
4.  lcp_crtpol2 --create --type list --pol list.pol --data list.data list_unsig.lst
  - copy and paste
./lcp_crtpol2 --create --type list --pol list.pol --data list.data list_unsig.lst
5.  cp list.pol /boot
  - copy and paste
cp owner_list.data /boot
```

Next create a verified Launch policy:

```
1.  tb_polgen/tb_polgen --create --type nonfatal vl.pol
  - copy and paste
../tb_polgen/tb_polgen --create --type nonfatal vl.pol
2.  tb_polgen/tb_polgen --add --num 0 --pcr 18 --hash image
    --cmdline "the command line from linux in grub.conf"
    --image /boot/vmlinuz-2.6.18.8
    vl.pol
  - copy and paste
../tb_polgen/tb_polgen --add --num 0 --pcr 18 --hash image --cmdline "root=UUID=cf6ae6b5-
abb5-4d5d-b823-bd798a0621de ro quiet splash $vt_handoff" --image /boot/vmlinuz-3.5.0-23-
generic vl.pol
3.  tb_polgen/tb_polgen --add --num 1 --pcr 19 --hash image
    --cmdline ""
    --image /boot/initrd-2.6.18.8
    vl.pol
  - copy and paste
../tb_polgen/tb_polgen --add --num 1 --pcr 19 --hash image --image /boot/initrd-3.5.0-23-
generic vl.pol

If any mle can be launched:
./lcp_crtpol2 --create --type any --pol any.pol

Define tboot error TPM NV index:
1.  lcptools/tpmnv_defindex -i 0x20000002 -s 8 -pv 0 -rl 0x07 -wl 0x07
    -p TPM-password
  - copy and paste
../lcptools/tpmnv_defindex -i 0x20000002 -s 8 -pv 0 -rl 0x07 -wl 0x07

Define LCP and Verified Launch policy indices:
1.  lcptools/tpmnv_defindex -i owner -s 0x36 -p TPM-owner-password
  - copy and paste
../lcptools/tpmnv_defindex -i owner -s 0x36 -p TPM-owner-password
2.  lcptools/tpmnv_defindex -i 0x20000001 -s 256 -pv 0x02 -p TPM-owner-password
  - copy and paste
../lcptools/tpmnv_defindex -i 0x20000001 -s 256 -pv 0x02 -p TPM-owner-password

Write LCP and Verified Launch policies to TPM:
(modprobe tpm_tis; tcsd;)
1.  lcptools/lcp_writepol -i owner -f [any.pol|list.pol] -p TPM-password
  - copy and paste
../lcp_writepol -i owner -f list.pol -p <ownerauth password>
2.  If there is a verified launch policy:
    lcptools/lcp_writepol -i 0x20000001 -f vl.pol -p TPM-password
  - copy and paste
../lcptools/lcp_writepol -i 0x20000001 -f vl.pol -p TPM-password
```

Use lcp_crtpollist to sign the list:

1. openssl genrsa -out privkey.pem 2048

2. openssl rsa -pubout -in privkey.pem -out pubkey.pem
3. cp list_unsig.lst list_sig.lst
4. lcp_crtpollist --sign --pub pubkey.pem --priv privkey.pem --out list_sig.lst

Use openssl to sign the list:

1. openssl rsa -pubout -in privkey.pem -out pubkey.pem
2. cp list_unsig.lst list_sig.lst
3. lcp_crtpollist --sign --pub pubkey.pem --nosig --out list_sig.lst
4. openssl genrsa -out privkey.pem 2048
5. openssl dgst -sha1 -sign privkey.pem -out list.sig list_sig.lst
6. lcp_crtpollist --addsig --sig list.sig --out list_sig.lst

# TPM 2.0 Installation and Operation

# Tboot

To install tboot, download the source code and ACM file from souceforge.

Copy the ACM file to the /boot directory.

In the directory with the untarred source type (as root):

```
make
make install
```

That's it. You will have to adjust your Grub configuration to user tboot. This is covered in a subsequent appendix.

Some notes on tboot's use of the TPM follow.

Tboot provides support to TPM2 module, and following command line option is used to select TPM2 extend policy.

extpol=agile|embedded|sha1|sha256|sm3|...

When "agile" policy is selected, ACM will use specific TPM2 commands to compute hashes and extend all existing PCR banks at the expense of possible performance loss.

For "embedded" policy, ACM will use algorithms supported by tboot to compute hashes and then will use TPM2_PCR_Extend commands to extend them into PCRs. If PCRs utilizing hash algorithms not supported by SW are discovered, they will be capped with "1" value. This policy when selected will ensure maximum possible performance at the expense of possible capping of some of the PCRs.

Other policy, like "sha1", "sha256", etc., only represent one single algorithm. It means tboot will use this algorithm to compute hash and use TPM2_PCR_Extend to extend it into PCRs.

PCR Usage --- Legacy PCR mapping

PCR 17 will be extended with the following values (in this order):
The values as documented in the MLE Developers Manual SHA-1 hash of: tboot policy control value (4 bytes) | SHA-1 hash of tboot policy (20 bytes): where the hash of the tboot policy will be 0s if TB_POLCTL_EXTEND_PCR17 is clear.
PCR 18 will be extended with the following values (in this order): SHA-1 hash of tboot (as calculated by lcp_mlehash), SHA-1 hash of first module in grub.conf (e.g. Xen or Linux kernel)
PCR * : tboot policy may specify modules' measurements to be extended into PCRs specified in the policy. The default tboot policy will extend, in order, the SHA-1 hashes of all modules (other than 0) into PCR 19.

Progress of the launch process is indicated via debug printk's using three different logging methods:

1. serial - logging is traced over a COM/serial port to a remote console
2. vga - logging is traced to the local screen
3. memory - logging is traced to a memory location

These three methods are not mutually exclusive - any combination can be enabled. Logging is enabled with command line parameters to tboot. The first parameter enables or disables logging levels (note that the default is all); any combination of "err", "warn", "info", "detail" can be used: loglvl=err,warn,info,detail|all|none

The next parameter is used to configure the various logging targets; any combination can be used (note that when the parameter is not set, serial is the default): logging=vga,serial,memory

If vga logging is set, the vga_delay parameter can be used to specify the number of seconds to pause after every screenful of output. It is specified as: vga_delay=<secs>

If serial logging is set, the serial port settings can be configured with the following parameters: serial=<baud>[/<clock_hz>][,<DPS>[,<io-base>[,<irq>[,<serial-bdf> [,<bridge-bdf>]]]]].

Tboot will copy and alter the e820 table provided by GRUB to "reserve" its own memory plus the TXT memory regions. These are marked as E820_UNUSABLE or E820_RESERVED so that the patched Xen code can prevent them from being assigned to dom0. The e820 table is not altered if the measured launch fails for any reason.

## Initramfs

Yours initramfs must contain the Cloudproxy root software and any shared libraries required by the kernel and the programs it starts with root privilege. We describe below how to do this.

For other (stacked) Cloudproxy hosts or applications, you have three choices:

1. Ensure all all programs are “statically linked” so you do not rely on “untrusted” libraries.
2. Ensure that only libraries in intiramfs are dynamically loaded with your application.
3. Control library loading from within your Cloudproxy application and make sure you load the library into memory and check it hash against a policy signed statement before linking to it.

Decompress and unpack the existing initramfs

```
cd /tmp
mkdir init
cd init
cp /boot/initramfs.img-`uname -r` initrd.gz
gunzip -c -9 initrd.gz | cpio -i -d -H newc --no-absolute-filenames
```

Copy the dynamic link libraries for the executables into the filesystem to the resulting directory.

Copy the runtime directory into the filesystem, and get ifconfig, too

```
mkdir -p home/jlm/jlmcrypt
cp -r /home/jlm/jlmcrypt/* home/jlm/jlmcrypt/
cp /sbin/ifconfig sbin/ifconfig
```

Change the initscript (init) to run dmcrypt and change the way the system disk is mounted.

vim init

Here is a simple script that works:

```
-- start simple script

#!/bin/sh

[ -d /dev ] || mkdir -m 0755 /dev
[ -d /root ] || mkdir -m 0700 /root
[ -d /sys ] || mkdir /sys
```

```
[ -d /proc ] || mkdir /proc
[ -d /tmp ] || mkdir /tmp
mkdir -p /var/lock
mount -t sysfs -o nodev,noexec,nosuid sysfs /sys
mount -t proc -o nodev,noexec,nosuid proc /proc
# Some things don't work properly without /etc/mtab.
ln -sf /proc/mounts /etc/mtab

grep -q '\<quiet\>' /proc/cmdline || echo "Loading, please wait..."

# Note that this only becomes /dev on the real filesystem if udev's
scripts
# are used; which they will be, but it's worth pointing out
if ! mount -t devtmpfs -o mode=0755 udev /dev; then
      echo "W: devtmpfs not available, falling back to tmpfs for /dev"
      mount -t tmpfs -o mode=0755 udev /dev
      [ -e /dev/console ] || mknod -m 0600 /dev/console c 5 1
      [ -e /dev/null ] || mknod /dev/null c 1 3
fi
mkdir /dev/pts
mount -t devpts -o noexec,nosuid,gid=5,mode=0620 devpts /dev/pts ||
true
mount -t tmpfs -o "nosuid,size=20%,mode=0755" tmpfs /run
mkdir /run/initramfs
# compatibility symlink for the pre-oneiric locations
ln -s /run/initramfs /dev/.initramfs

/sbin/ifconfig lo 127.0.0.1
# can set up other networks here as needed, e.g., on eth0

# mount /boot as a place to put keys between reboots (e.g., for
tcService.exe)
mkdir /boot
mount /dev/sda1 /boot

/bin/busybox sh

-- end simple script

  - untested:
      swapoff -a
      cryptsetup [-c aes -h sha256] -s 128 -d /dev/urandom create swap
/dev/sda1
      mkswap /dev/mapper/swap
      swapon /dev/mapper/swap
```

### Put initramfs back together

```
find . | cpio -H newc -o|gzip -9 > ../initrd.img-new
```

Copy it to the boot directory

```
sudo cp initrd.gz /boot/initrd.img-staticLinux
```

Change /etc/grub.d to use this new initramfs.

To configure Linux to boot by mounting an initramfs image internally, set the following items on the kernel config:

```
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE=<Initramfs cpio image path>
```

After making these changes to the settings, re-compile the kernel. Finally, the kernel can be booted as follows (for instance, we report a boot on a mb442 by using a "vmlinux" with the above settings enabled):

```
host% st40load_gdb -t stmc:mb442:st40 -c sh4tp -b vmlinux mem=64m
```

No other parameters have to be passed to st40load_gdb to boot with initramfs, as the kernel mounts that image automatically.

To boot Linux by using the initramfs image, create a typical ext2 (or ext3) filesystem structure in which the only difference with respect to the standard ext2/3 root filesystem used in SysV, or busybox, is related to the init file.

Usually, the init file is placed under /sbin (for both SysV and Busybox). In the initramfs rootfs image, the init file has to be placed under the root, /, as the kernel tries to execute /init instead of /sbin/init in case the initramfs image was mounted after boot. A typical initramfs filesystem structure is as follows:

target% ls

bin dev etc home include init lib mnt proc sys sbin tmp usr var

The tool mkshinitramfs has been created to improve the process of initramfs rootfs image creation. The current version of the tool is 1.0. To get information about the tool's parameters, type the following command line:

```
host# ./mkshinitramfs -h
mkshinitramfs-1.0,
usage: mkshinitramfs [-h|-H   Help]
       mkshinitramfs [-o|O:  <Initramfs CPIO image with switch option
to HDD(/dev/sda1)>]
       mkshinitramfs [-s|S:  <Initramfs CPIO image with shell on
Initramfs>]
       mkshinitramfs [-p|P:  <target rootfs path> | default:
/opt/STM/STLinux-2.3/devkit/sh4/target/]
```

```
        mkshinitramfs [-k|K:  <SH kernel version>  | default:
2.6.23_stm23_0121]
        mkshinitramfs [-b|B:  <used board> | default: mb442 (STi710x
based board)]
```

mkshinitramfs must be executed as "root" on the host machine in which a full target root filesystem has previously been installed. It is configured to use default parameters, as listed in the output from the -h option. To change any of the parameters, apply the command line options listed in the help.

The tool mkshinitramfs creates a short initramfs. If invoked with the -s option, the generated initramfs is the final rootfs. If invoked with the -o option, the generated initramfs is a temporary rootfs to be switched to the real one on the HDD.

The script mkshinitramfs internally defines some variables that are useful when defining which files, libraries, scripts and tools are to be placed on the initramfs image.
The variables are:

```
$DIRS
$BINFILES
$SBINFILES
$ETCFILES1
$USRBINFILES
```

Version 1.0 of the mkshinitramfs tool can be downloaded from mkshinitramfs tool as a compressed tar.gz file.

If the initramfs rootfs is created to be used temporarily before switching to the real target rootfs on a hard disk drive, a setup script is needed, which is then passed to the chroot in order to setup the final real rootfs properly. This script loads the user space services needed at runtime, such as klogd, udev, D-Bus, and so forth.

If the -o option is used, mkshinitramfs generates a CPIO image in which the following command line is inserted after the kernel mounts the initramfs:

```
mount /dev/sda1 /mnt
chroot /mnt /etc/init.d/rcS-initramfs
```

# Dmcrypt

Dmcrypt is a standard Linux component.

fdisk to make partitions (/dev/sda)
mkfs.ext2  to make file system
change to single user mode
grub-install root-directory=/dev/sda
CRYPTDISKS_ENABLE=Yes in /etc/defaults/cryptdisks

/etc/crypttab
swap /dev/hda1 /dev/urandom swap
/etc/fstab
/dev/swapper/swap none
swap sw,pri=1 0 0

For swap:

swapoff -a
cryptsetup [-c aes -h sha256] -s 128 -d /dev/urandom create swap /dev/sda1
mkswap /dev/mapper/swap
swapon /dev/mapper/swap

Shell script prototype

```
#
#   For debian, first do the following:
#
#   Change to single user mode
#       cat /proc/swaps   (to find out swap device)
#       /sbin/runlevel    (to find runlevel)
#       /sbin/telinit 1   (to go single user)
#       ctrl-alt-f7 puts Linux into console mode.
#
#   /etc/defaults/cryptdisks
#       CRYPTDISKS_ENABLE=Yes
#   /etc/crypttab
#       swap /dev/sda5 /dev/urandom swap
#   /etc/fstab
#       /dev/mapper/swap none
#       swap sw,pri=1 0 0
#   /etc/crypttab
#       swap /dev/sda5 /dev/urandom cipher=aes-cbc-essiv:sha256,size=256,hash=sha256,swap
#   /etc/fstab
#       /dev/mapper/cryptoswap none swap sw 0 0
#
#   /dev/sda5 is swap device
#
# Ref: The whole disk Nov, 2006, Linux-magazine.com, Michael Nerb
```

```
#
swapoff -a
cryptsetup -c aes -h sha256 -s 128 -d /dev/urandom create swap /dev/sda5
mkswap /dev/mapper/swap
swapon /dev/mapper/swap
#
# To remove:
#   swapoff /dev/mapper/swap
#   cryptsetup [luks] remove swap
```

# Grubby, Grub, Grub

The new tboot module must be added as the 'kernel' in the grub.conf file. The existing 'kernel' entry should follow as a 'module'. The SINIT ACM module must be added to the grub.conf boot config as the last module, for example:

```
       root (hd0,1)
            kernel /tboot.gz logging=serial,vga,memory
            module /vmlinuz-2.6.18 root=/dev/VolGroup00/LogVol00 ro
            module /initrd-2.6.18.img
            module /Q35_SINIT_17.BIN
```

The kernel module is tboot. The next is the Linux (possibly KVM enabled) kernel. Next is the initramfs and finally the ACM module.

GRUB2 does not pass the file name in the command line field of the multiboot entry (module_t::string). Since the tboot code is expecting the file name as the first part of the string, it tries to remove it to determine the command line arguments, which will cause a verification error. The "official" workaround for kernels/etc. that depend on the file name is to duplicate the file name in the grub.config file like below:

```
    menuentry 'Xen w/ Intel(R) Trusted Execution Technology' {
            recordfail
            insmod part_msdos
            insmod ext2
            set root='(/dev/sda,msdos5)'
            search --no-floppy --fs-uuid --set=root 4efb64c6-7e11-482e-8bab-
07034a52de39
            multiboot /tboot.gz /tboot.gz logging=vga,memory,serial
            module /xen.gz /xen.gz iommu=required dom0_mem=524288
com1=115200,8n1
            module /vmlinuz-2.6.18-xen /vmlinuz-2.6.18-xen
root=/dev/VolGroup...
            module /initrd-2.6.18-xen.img /initrd-2.6.18-xen.img
            module /Q35_SINIT_17.BIN
       }
```

## Building and running KVM

Here is a quick description of how to build the kernel for host and guest.

variables: $KERNEL is the kernel source directory (e.g., /usr/src/linux-3.11.0), $CODE is the fileProxy Code directory (e.g.,~/src/fileProxy/Code)

```
export KERNEL="/usr/src/linux-lts-quantal-3.5.0"
export CODE="/home/jlm/fpDev/fileProxy/Code"
```

- Make sure you have all the prerequisites for building the kernel
- get the kernel source and make its root directory be $KERNEL
- navigate to ${CODE}/kvm/host-kernel
- execute the commands
  - ./apply-patches.sh $KERNEL
  - ./copy_to_kernel.sh $KERNEL

make menuconfig
  disable loadable module support
  edit config CONFIG_ACPI=n

The apply-patches.sh command might fail, depending on the vintage of the kernel you have. If the patches fail, they are fairly easy to fix by hand. You can look at the patches to see what they do:
  - the Makefile in ${KERNEL}/drivers/misc needs to have three extra lines, one obj-y += DIR for each of the three directories we added (*tcio*)
  - the Makefile in ${KERNEL}/arch/x86/kvm needs to add vmdd.o to the dependencies of kvm.o
  - ${KERNEL}/arch/x86/kvm/x86.c needs to have the lines added from the patch in ${CODE}/kvm/host-kernel/arch/x86/kvm/x86.c.patch (just acouple of lines)

- copy a working config over to $KERNEL as .config.  This is more important than it may seem at first glance.  Often dependencies in make menuconfig make getting an appropriate configuration (e.g.-one that CONFIG_INTEL_TXT=y) difficult.  A sample .config file (for Linux 3.5.7) is in config.sample in this directory. NOTE: If you're building in the VM make sure you copy a .config from a working VM kernel.
- run
  - yes "" | make oldconfig
  - yes "" | make localmodconfig
  - make (with a -j factor appropriate for your machine)
  - sudo make modules_install
  - sudo make install
- and you should have a copy of the new kernel in /boot, ready to go.
- update the tboot script for this kernel and try to tboot it.

# Building and running KVM VMs

Make sure your VM has at least 20MB of disk

1. Install via virt-manager a fresh VM from a CD.
2. SSH the jlmcrypt directory into the VM.
   apt-get install openssh-server
   sudo service ssh start
   sudo stop ssh
   sudo start ssh
   sudo restart ssh
   sudo status ssh
   ssh jlm@192.168.122.244
   sftp
   put jlmcrypt.tar .

3. Save the VM Image in virtmanager. You can find the IP address to SSH to by going into the VM and typing ifconfig -a.

4. Save the original libvirt xml to a safe place. For example,
   virsh dumpxml KvmTestGuest > /home/jlm/jlmcrypt/vms/KvmTestGuestImage.xml

5. Copy the kernel-image file and initramfs file for the CloudProxy partition to the place you want to save them. In this example, I use /home/jlm/jlmcrypt/vms/kernel and /home/jlm/jlmcrypt/vms/initram.

6. Change the boot options for your vm (in virt-manager, for example) so that it does the direct kernel/initram boot using the files named in step 6.

7. Save the new libvirt xml to a safe place. For example,
   virsh dumpxml KvmTestGuest > /home/jlm/jlmcrypt/vms/KvmTestGuestBoot.xml.

8. Copy the direct boot xml into a template file that you will reference
   when you tcLaunch the VM. For example, cp /home/jlm/jlmcrypt/vms/KvmTestGuestBoot.xml /home/jlm/jlmcrypt/vms/KvmTestGuestTemplate.xml

9.Edit the template file.
   Replace the value of the name tag (which should have the name of
   vm originally) with %s. For example,
      <name>%s</name>
   Replace the kernel and image tag values with %s, getting
      <kernel>%s</kernel>
      <initrd>%s</initrd>

Replace the disk source file value with %s, getting:

```
<disk type='file' device='disk'>
   <driver name='qemu' type='raw'/>
   <source file='%s'/>
   <target dev='hda' bus='ide'/>
   <address type='drive' controller='0' bus='0' unit='0'/>
</disk>
```

Running the newly-installed guest

sudo /usr/local/kvm/bin/qemu-system-x86_64 vdisk.img -m 384

Download and install KVM: You can use the package included in Ubuntu 7.10, just execute: apt-get install kvm You can also download the last release of KVM and compile it.

You also need the linux-headers packages for the Kernel you are using.

First of all, we need the KVM package. You can find the actual release at http://sourceforge.net/projects/kvm/. Download it to your server:

```
wget link/to/file /usr/src/
```

Now unpack the content.

```
cd /usr/src/
tar -xzf file.tar.gz
cd folder/to/kvm
```

Before compiling KVM make sure that your CPU supports virtualization and if that option is activated in the BIOS. Compile KVM.

```
./configure
make
make install
```

Now load the KVM modules:
If you have an Intel CPU load:

```
modprobe kvm
modprobe kvm_intel
```

If you have an AMD CPU load:

```
modprobe kvm
modprobe kvm_amd
```

Creating the network bridge First, install bridge-utils: apt-get install bridge-utils

If you want to connect to your virtual machine later, you need to set up a network bridge. Edit /etc/network/interfaces like this (make sure to insert your public IP Address):

```
auto lo
iface lo inet loopback
auto br0
iface br0 inet static
address xxx.xxx.xxx.xxx
netmask xxx.xxx.xxx.xxx
gateway xxx.xxx.xxx.xxx
bridge_ports eth0
bridge_stp off
bridge_maxwait 5
```

Also edit /etc/qemu-ifup like this:

```
#!/bin/sh
/sbin/ifconfig $1 0.0.0.0 promisc up
/usr/sbin/brctl addif br0 $1
sleep 2
```

I recommend restarting your server at this point. If you are able to reconnect, your bridge is working.

Creating a virtual disk for a virtual machine & install a system into it. Create an .img file for installation. This file acts as a virtual disk for your vm.

## Host Certificates